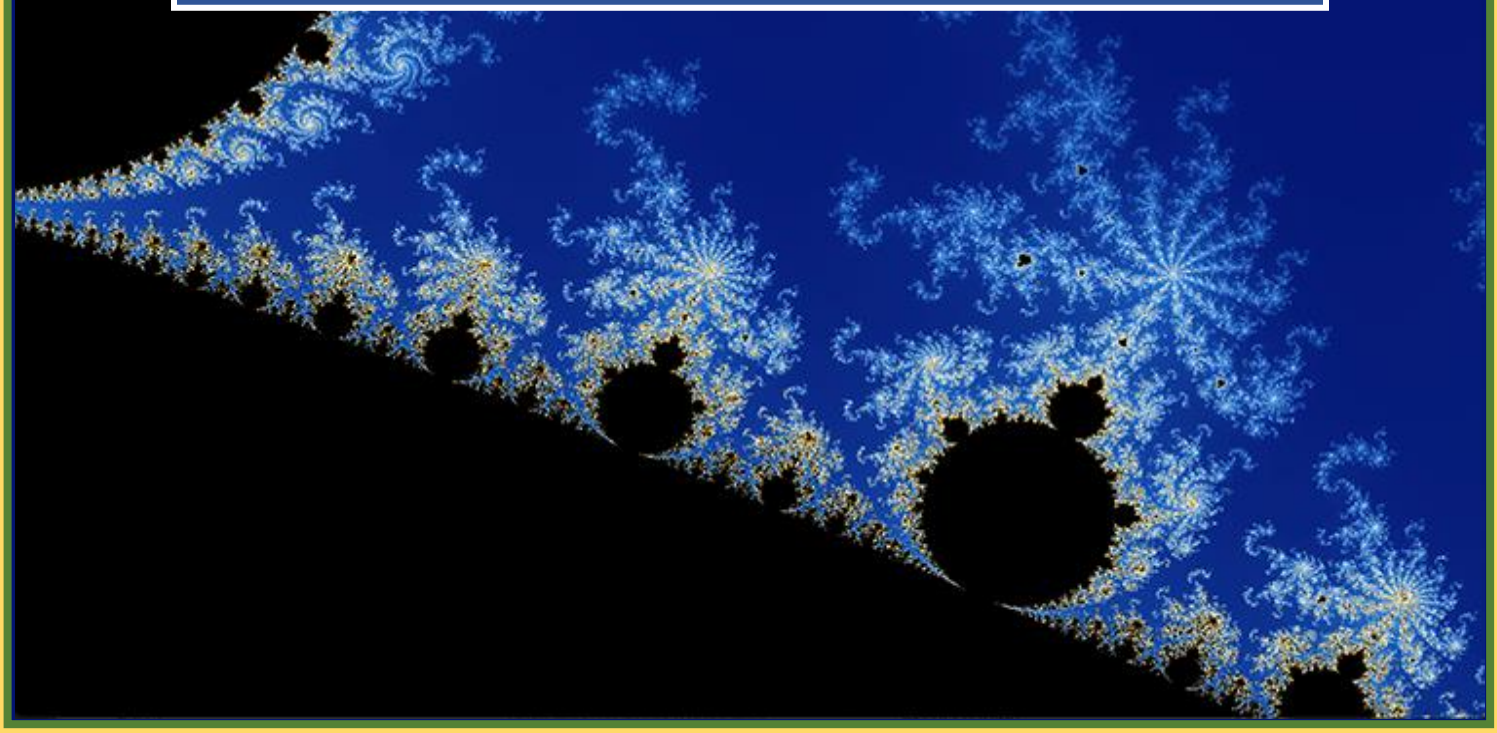
A Summary and Extra Notes On Chapter 3



Order of topics that are going to be covered in this file:

- **A small mistake on this chapter**
- **Block of code in curly braces**
- **What happens if we declare a variable inside the loop block?**
- **An almost deep look at the compiler**
- **i++ or ++i ? Which one?**
- **The comma operator**
- **Difference between '\n' and endl**
- **enter character**

A small mistake on this chapter

In this chapter there's a small mistake that I also investigated as well. The mistake appears on section 3.3 called "Input and Type". Before further explanation, please take a look at the code presented:

```
// read name and age
int main()
{
std::cout << "Please enter your first name and age\n";
std::string first_name = "???"; // string variable
int age = -1; // integer variable
std::cin >> first_name; // read a string
std::cin >> age; // read an integer
std::cout << "Hello, " << first_name << " (age " << age << ")\n";
}
```

This section asks you to try out different types of input. Two main inputs being tried out and their outputs are as follows:

1. Carlos 22

```
Please enter your first name and age
Carlos 22
Hello, Carlos (age 22)

Process returned 0 (0x0)   execution time : 3.583 s
Press any key to continue.
```

2. 22 Carlos

```
Please enter your first name and age
22 Carlos
Hello, 22 (age 0)

Process returned 0 (0x0)   execution time : 2.705 s
Press any key to continue.
```

The first case is fine. However, on the second case the book mentions that instead you get "Hello, 22 (age -1)" which is not the case. I asked about this on r/cpp_question and here are some of the answers I've received:

" The std::istream::operator>> performs a call to std::num_get::get. The documentation of the later explicitly says: If the conversion function fails to convert the entire field, the value 0 is stored in v."

" if the stream fails it will set the int to 0 while the book suggests it leaves the previous value. "

You can read more about this from this link below:
https://www.reddit.com/r/cpp_questions/comments/thsi9g/im_not_getting_the_expected_output/

Block of code in curly braces

A group of statements wrapped in between the curly braces ("{}") is called a block of code.
IMPORTANT NOTE: If you have more than one statement, you must wrap your code in curly braces. This applies to if conditions, loops and functions. Here is an example showing a block of code:

```
for(char c:secretMessage){
    size_t position = alphabet.find(c);
    if (position != string::npos){
        char newChar {key.at(position)};
        encryptedMessage += newChar;
    }
    else
        encryptedMessage += c;
}
```

What happens if we declare a variable inside the loop block?

I would like to demonstrate this section with a very simple example:

```
#include <iostream>

int main() {
    for(int i{0}; i<5; i++){
        int number {0};
        number += i;
    }
    std::cout << number << '\n'; //error: 'number' was not declared in this scope
    return 0;
}
```

As you can see in this example, we get an error as specified. This is a concept for variables. It's called "Local" or "Global" variables.

Local variables: A rule of thumb is that, any variable declared inside a block of code is considered a local variable. This means that the variable is only accessible within that block of code or in other words, within that scope. After that block of code is executed completely, those variables within that block will be deleted and cannot be accessed anymore.
Global Variables: On the other hand, global variables are declared outside blocks of code and their values can always be accessed until the end of the program.

NOTE: A variable can at the same time be local and global. For example, a variable declared inside the main function is considered global to other blocks of code within main but local to only main. Let me show you an example:

```
#include <iostream>
void display(){
    std::cout << number << '\n'; //error: 'number' was not declared in this scope
}
int main() {
    int number {13};
```

```
    display();
    return 0;
}
```

As you can see, number is only global inside main. However, if we do this we can see our code works:

```
#include <iostream>
void display(){
    int number {13};
    std::cout << number << '\n';
}
int main() {
    display();
    return 0;
}
```

This time number is local to the "display" function. The problem is that you can't access that within main anymore. To solve this problem, you can provide an argument within your function:

```
#include <iostream>
void display(int num){
    std::cout << num << '\n';
}
int main() {
    int number {13};
    display(number);
    return 0;
}
```

So the concept as you can see is pretty straightforward. In our case of having a loop and declaring a variable within the loop block, what happens is that at the end of the execution of the loop, that variable will be deleted and can't be accessed anymore.
This concept of local variables is useful in terms of memory management. This helps with the deletion of variables that are not going to be used anymore throughout the program and a freer memory in return resulting in a better performance.

An almost deep look at the compiler

The compiler comes with some interesting flags. In this section I'm going to cover some of the general flags that we all use daily.

`-Wall`: This flag is responsible for showing you warnings generated when compiling your programs. By default, most IDEs call with this flag.

`-Wextra`: This enables some extra warning flags that are not enabled by -Wall.

`-std=c++2a`: With this flag, you can specify which version of C++ to use. For example: -std=c++20 uses C++20.

`-pedantic`: Issues all the warnings demanded by strict ISO C and ISO C++; rejects all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++.

The list is very long. There are two links that I highly recommend checking them out.

https://stackoverflow.com/questions/3375697/what-are-the-useful-gcc-flags-for-c

https://www.reddit.com/r/cpp/comments/grpux6/which_gcc_flags_do_you_use_often/

There is also documentation for gcc which I recommend just taking a very brief look at them.

https://gcc.gnu.org/onlinedocs/gcc/

i++ or ++i? Which one?

There's been a discussion about the difference regarding i++ or ++i. Let's show this with a common example that I saw researching about this:

```
#include <iostream>

int main() {
    int i {1};
    int j {++i};
    std::cout << i << '\n' << j << '\n';
    return 0;
```

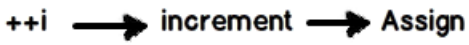
```
Output:
i = 2
j = 2
```

```
#include <iostream>

int main() {
    int i {1};
    int j {i++};
    std::cout << i << '\n' << j << '\n';
    return 0;
```

```
Output:
i = 2
j = 1
```

The explanation can be summarized into this figure:



In general, ++i will increment the value of i, and then return the incremented value. i++ will increment the value of i, but return the original value that i held before being incremented.
It's best to go with ++i in most cases. ++i seems more common, perhaps because that is what is used in K&R (Or C programming language).

You can read more about it in here:
https://stackoverflow.com/questions/24853/what-is-the-difference-between-i-and-i

There's also this discussion whether i++ or ++i impact performance. In general, there's not much of a difference mostly because of the optimization that the compiler does. You can read more about it here: https://stackoverflow.com/questions/24886/is-there-a-performance-difference-between-i-and-i-in-c

In my opinion, best case to go with ++i.

The comma operator

The comma operator has the lowest precedence of all C/C++ operators. Therefore it's always the last one to bind to an expression. For example:

```
a = b, c;
```

is equal to: (a=b), c;

Overall, it's best to avoid this and instead use semicolon to terminate your statements as it's clearer and more understanding. However, I'm going to show an example involving an if condition:

```
#include <iostream>
int main() {
    int age{23};
    if(std::cin >> age, age>0)
        std::cout << "David is " << age << " years old.";
    return 0;
}
```

In this example, the if condition receives your input and also checks if age is bigger than zero or not. This is equivalent to this:

```
#include <iostream>
int main() {
    int age{23};
    if(std::cin >> age && age>0)
        std::cout << "David is " << age << " years old.";
    return 0;
}
```

You can read more about the comma operator here:

https://stackoverflow.com/questions/54142/how-does-the-comma-operator-work-and-what-precedence-does-it-have

https://en.wikipedia.org/wiki/Comma_operator

Difference between '\n' and endl

There has been different variations between programmers. Some programmers prefer to use '\n' instead of endl. The thing is there must be a difference between two applications. I did some research about this and here's something that I found:

"

The main difference is that std::endl flushes the output buffer, and '\n' doesn't. If you don't want the buffer flushed frequently, use '\n'. If you do (for example, if you want to get all the output, and the program is unstable), use std::endl.

"

Another interesting response I found is this:

"

When you read or write, you may be reading or writing to memory, not to a file (which could be a file on disk, a network socket, a pipe, etc). The idea behind buffering is that some IO can be made more efficient by writing larger batches of data at a time less frequently. Buffering mechanisms will write the buffer contents once the buffer is full. This is called flushing, like a toilet - when your buffer is full of shit, you flush it.

You can also flush manually, and that's what std::flush is for. The reason to do this is for an interactive program, when you want responsiveness, when you're writing a little bit of data infrequently, so you may never fill the buffer so it flushes automatically. You may be writing to a device that expects data in particular message sizes or given a certain promptness. You may be writing data to a target that is a big expensive computation, so you want it to get started sooner than later. If your program is a filter, part of a process pipeline at a console window, you may want to consider setting std::unitbuf, so the down-pipe programs get data sooner, and ultimately the terminal is more responsive to the user.

Typically, you never have to flush manually. Don't do it until you hit that wall and you know you have to do it.

"

Personally, I think '\n' is better in most cases and in cases where it fills necessary to flush the stream, it can be done manually. You can read more about this here:

https://www.reddit.com/r/cpp_questions/comments/iuczni/what_does_flushing_the_output_buffer_mean_in_cpp/

https://stackoverflow.com/questions/213907/stdendl-vs-n

---

enter character

There was a question that was asked in the discussions in the discord server. It goes something like this: " I'm trying to make a "press enter to exit" situation, but I got stuck on wait type of variable to use for it. I tried to think what kind of character would "enter" be, or if it would be, not a character, but an interaction of some sort. "

I did some research regarding this. General idea that came to my mind was this solution:

```
#include <iostream>
int main() {
    char c{'\n'};
    std::cout << "Press Enter To Exit: ";
    std::cin >> c;
    if(c == '\n')
        return 0;
}
```

The problem is that cin doesn't take '\n' as a character. It takes only ASCII characters. A solution that I found involves using cin.get():

```
#include <iostream>
int main(){
    std::cout << "Press Enter To Exit: ";
    std::cin.get();
    return 0;
}
```

You can read more about it here:

https://cplusplus.com/forum/beginner/2624/

https://stackoverflow.com/questions/42818899/detecting-enter-key-in-c